electronics

MDPI

Article

Statement-Grained Hierarchy Enhanced Code Summarization

Qianjin Zhang, Dahai Jin *, Yawen Wang and Yunzhan Gong

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China; js_0102@bupt.edu.cn (Q.Z.)
* Correspondence: jindh@bupt.edu.cn

Abstract: Code summarization plays a vital role in aiding developers with program comprehension by generating corresponding textual descriptions for code snippets. While recent approaches have concentrated on encoding the textual and structural characteristics of source code, they often neglect the global hierarchical features, causing limited code representation. Addressing this gap, our paper introduces the statement-grained hierarchy enhanced Transformer model (SHT), a novel framework that integrates global hierarchy, syntax, and token sequences to automatically generate summaries for code snippets. SHT is distinctively designed with two encoders to learn both hierarchical and sequential features of code. One relational attention encoder processes the statement-grained hierarchical graph, producing hierarchical embeddings. Subsequently, another sequence encoder integrates these hierarchical structures with token sequences. The resulting enriched representation is then fed into a vanilla Transformer decoder, which effectively generates concise and informative summarizations. Our extensive experiments demonstrate that SHT significantly outperforms state-of-the-art approaches on two widely used Java benchmarks. This underscores the effectiveness of incorporating global hierarchical information in enhancing the quality of code summarizations.

Keywords: source code summarization; code representation learning; code static analysis; program comprehension; Transformer

Citation: Zhang, Q.; Jin, D.; Wang, Y.; Gong, Y. Statement-Grained Hierarchy Enhanced Code Summarization. *Electronics* **2024**, *13*, 765. https://doi.org/10.3390/electronics13040765

Academic Editors: Hsi-Min Chen and Shang-Pin Ma

Received: 23 January 2024
Revised: 10 February 2024
Accepted: 14 February 2024
Published: 15 February 2024

1. Introduction

As software continues to expand in both size and complexity, developers dedicate approximately 90% of their efforts to software maintenance tasks (version iteration and bug fixing) throughout the entire software development lifecycle [1]. Program comprehension is crucial in both software development and maintenance, and providing natural language summaries for the source code significantly eases the burden on developers [2]. Source code summarization involves the creation of easily understandable summaries that explain a program's functionality [3]. Nevertheless, manually crafting code summaries is a laborious and time-consuming task. Hence, source code summarization, which automates the generation of concise program descriptions, holds great significance [4].

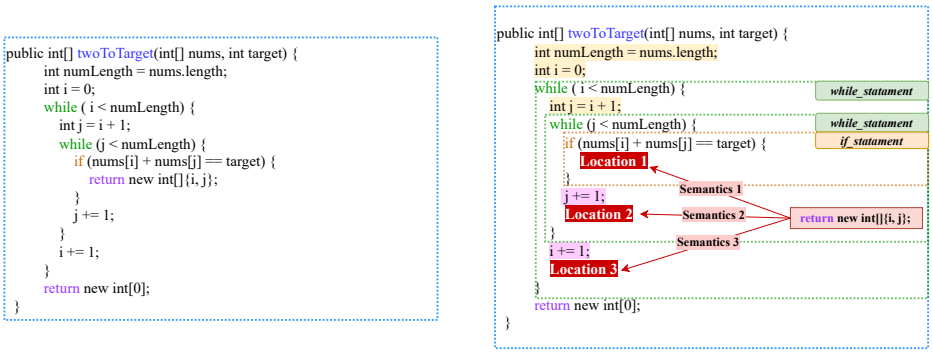
Code representation is a major topic in ML-based code summarization, and code usually is converted into sequence or structure format for code semantic embedding. Sequence-based methods of code summarization leverage code sequential information (i.e., text, token sequences) for semantic representation and benefit from capturing the contextual and sequence order of code snippets. Previous approaches [5,6] treated code as sequential tokens and employed well-established sequence-to-sequence models for summarization tasks. Although code snippets do exhibit certain similarities to conventional texts, they encompass more complex and explicitly structured information [7].

Abstract syntax trees (ASTs) and graph representations (i.e., Control Flow and Program Dependency graphs) offer structural representations of source code snippets. Structure-based approaches either utilize the ASTs to precisely represent the structural and grammatical features of code or employ graph representation to obtain additional code structures, like control and data dependencies. AST-based methods directly encode original ASTs [1,8,9]

or preprocessed ASTs, including flatten ASTs [10], paths [11], and subtrees of AST, using Recurrent Neural Networks (RNN) for code summarization generation. Graph-based methods represent the program as a graph in acquiring a deeper code semantic comprehension [12]. Many approaches adopt Graph Neural Networks (GNN) to capture code graph representations [13,14] or employ a Transformer model to encode AST-based graphs for summary generation [15,16]. However, these methods focus on modeling code structures but they often leverage limited structural information and omit the hierarchical structure.

The above methods for code comprehension predominantly focus on token-level granularity and structural information among tokens. However, in real-world scenarios, proficient programmers usually focus on understanding each statement within code snippets and leverage the hierarchical relationships among these statements to summarize the overall functionality of the code. Typically, each statement in a code snippet functions as a tiny functional component (i.e., loop, invocation). The hierarchical information among these statements refers to the logical relationships among these functional components, which are crucial in comprehending the overall purpose and functionality of the code.

For example, Figure 1 illustrates a code snippet in statement-level granularity for code understanding. The *twoToTarget* method contains five types of statements, including *methodDeclaration_statement*, *Assignment_statement*, *while_statement*, *if_statement*, and *return_statement*. Each statement represents a tiny function point. Through understanding the elements and logic among function points, we can roughly summarize the code as "*return the numbers of elements in source list equal to value*". The type of statement impacts all token nodes within the statement, as illustrated in Figure 1b. For instance, tokens of *while_statement* may be exacted more than once; tokens of *if_statement* may or may not be exacted. Meanwhile, the global hierarchical information of the statement is also related to the semantics of the statement. Figure 1b shows an instance in which a small change in the statements' position affects the program's output. The location of "*return new int[]{i, j}*" will completely change the result of *return_statement*. Locating the statement in the block structure will help the model to better determine the function of the source code. Meanwhile, Zhang et al. [17] observe that the functionality of the program is closely related to the global hierarchy via a statistical experiment on a code classification task.



(**a**) Original Code (**b**) Hierarchical Code

Figure 1. An illustrative example of the statement-grained hierarchy information in source code. (**a**) Original code contains several statements. (**b**) The statement type and hierarchical location affect the operational semantics of tokens in the statement.

Driven by the analysis outlined above, we propose unifying the hierarchical information of statements with token sequences for source code summarization. Initially, we employ the comma and grammar parser to segment the code snippets into individual statements. Subsequently, we construct a statement-grained semantic graph that contains both the hierarchical relationships among statements and statement type information, serving as the code structural skeleton. Drawing inspiration from [18], Natural Language Processing (NLP) techniques usually leverage the Transformer-based model to encode tokens, tokens' tags, and positions to comprehend the semantics of natural language. Programming lan-

guages, while sharing similarities with natural languages, exhibit more complex structures and adhere to stricter syntax rules. Therefore, we attempt to treat the statement-grained semantic graph, along with the syntax of tokens, as a representation of the global hierarchical position information and tags of code tokens. The aggregation of code tokens' positions, tags, and text is then fed into the sequence Transformer for effective code summarization.

To validate the effectiveness of the above-mentioned code semantic representation, we propose a novel framework, the statement-grained hierarchy Transformer (SHT). This framework is designed to capture the hierarchical characteristics of statements and aggregate statement-grained hierarchical features with sequential syntax and tokens, thereby generating concise natural language descriptions of code functions. The SHT model comprises two pivotal components: a Transformer-based hierarchy encoder, to learn the representation of statement-grained hierarchical information, and a Transformer-based sequence encoder for the aggregated code semantic representation. We conducted an empirical study to investigate the impact of the statement-grained hierarchical structure on code summarization, and the experimental results demonstrate the effectiveness of statement-grained hierarchical information. The SHT was rigorously evaluated on two public datasets for source code summarization tasks, where it surpassed previous works and achieved new state-of-the-art results. Additionally, ablation studies and human evaluation were conducted to illustrate the efficiency of the proposed method. In summary, our work makes the following contributions to the field.

- We are the first to explore the use of a statement-grained hierarchy graph for the extraction of global hierarchical structural properties. This graph is integrated with the code token sequence to represent code semantics for source code summarization.
- We propose a novel model, SHT, which incorporates the statement-grained hierarchy graph and token sequence to generate code summaries. This model uniquely combines two encoders for sequence and graph learning within the Transformer framework.
- Our approach is evaluated on two source code summarization benchmark datasets against baseline models, surpassing previous works and achieving new state-of-the-art results. Additionally, the ablation study and human evaluation further validate our strategy for code comprehension in code summarization.

The remainder of this paper is organized as follows. Section 2 reviews the background knowledge of the work. Section 3 defines the problem formulation of code summarization. Section 4 presents our proposed methodology. Section 5 details the experimental setup and discusses the results. Section 6 describes the related works, followed by the limitations of our work in Section 7. Finally, Section 8 concludes the paper and outlines directions for future research.

2. Background

In this section, we introduce the background knowledge of the proposed approach, including the Transformer model architecture and relational attention.

2.1. Transformer

Transformer [19] is a deep self-attention network that was initially proposed for neural machine translation and has demonstrated its powerful representation capabilities in many NLP applications [20,21]. The architecture of Transformer consists of multiple stacked encoder and decoder layers. Within each encoder stack, the inputs first pass through a self-attention sublayer and then are fed into a position-wise feed-forward sublayer, followed by layer normalization. The decoder features a similar structure to the encoder but includes a cross-attention layer. These layers are strategically positioned between the encoder's sublayers to enable the decoder to focus on relevant parts of the input sequence. Both the encoder and decoder employ residual connections and layer normalization to facilitate the flow and integration of information across their respective sublayers.

2.2. Self-Attention

Self-attention is the key component of the Transformer model. The attention mechanism can be formulated as querying a dictionary composed of key–value pairs. For a single head of attention, the matrix form equation is typically represented as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V \quad (1)$$

Here, the $Q(Query), K(Key)$, and $V(Value)$ metrics are derived from the inputs, and d represents the dimension of the vectors, which helps in scaling the dot products. The attention mechanism then computes the dot products of the query with all keys, scales these values, and applies a softmax function to obtain the weights on the values. The idea behind attention is that not all tokens in a sequence are of equal importance to the final output prediction. The attention mechanism tries to learn the most important words in a sequence. As each word in a sentence simultaneously flows through the encoder and decoder stack, the model does not inherently understand the order of words. To compensate for this, position embeddings are added to each word embedding.

2.3. Relation-Aware Self-Attention

Recently, there has been a surge in research that adopts the Transformer model as the backbone for enhanced source code representations [6,17]. This trend includes adapting the Transformer, originally designed for processing plain text, to handle structural data (graphs and trees) [15,16,22]. These structure-aware transformer models encode structures by introducing soft inductive biases. Relational attention [23,24], extended from self-attention, is widely applied in structure-aware Transformer. Comparing self-attention project QKV vectors from each node vector, relational attention's central innovation, as depicted in Figure 2, involves conditioning the QKV vectors on the directed edge e_{ij} between the nodes, which concatenates the edge vector e_{ij} with each node vector before the linear transformers. The process can be formally described as

$$q_{ij} = [n_i, e_{ij}]W^Q \qquad k_{ij} = [n_i, e_{ij}]W^K \qquad v_{ij} = [n_i, e_{ij}]W^V$$

where each weight matrix $W \in \mathbb{R}^{(d_n+d_e)d_n}$, and d_e is the edge vector size.

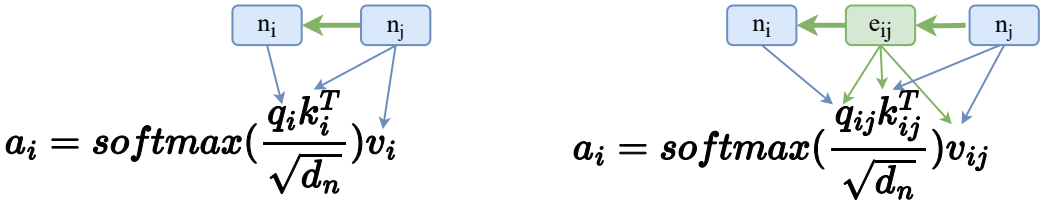


Figure 2. On the left is the standard Transformer attention computing QKV among node vectors. On the right is relational attention computing the edge vector as well.

3. Problem Formulation

In our work, we aim to generate readable and concise source code descriptions. We formalize the task as a supervised natural language generation problem. Formally, let D denote a dataset containing a set of programs C and their associated summarizations Z, given source code $c = (x_1, x_2, \ldots, x_n)$ from C, where n denotes the code sequence length. Furthermore, we leverage relational attention to extract the global hierarchical features of code sequence c. Finally, the aggregation of textual and hierarchical features is fed to the model to generate the code summary $\tilde{z} = (y_1, y_2, \ldots, y_m)$ by maximizing the conditional likelihood: $\tilde{z} = argmaxp(z|c)$ (z is the corresponding summary in the summary sets in Z).

4. Proposed Approach

This study introduces a novel code summarization model named SHT, which is designed to process both the code sequence and hierarchy information. The comprehensive pipeline of our approach is depicted in Figure 3. We initiate the process by extracting three-level semantic features from the ASTs. This feature set encompasses a statement-grained hierarchical graph, token sequence, and syntax sequence. Subsequently, the hierarchical graph is processed through a relational attention encoder block, yielding hidden vector representations of statement nodes. These representations encapsulate global hierarchical information and statement types. Following this, the aggregated hidden states of statement nodes, along with the embeddings of token and syntax sequences, are input into a Transformer-based sequence encoder. This encoder is employed to acquire the final semantic representation of the source code. Lastly, the resulting semantic representation is fed into a basic Transformer decoder to generate a concise summary.

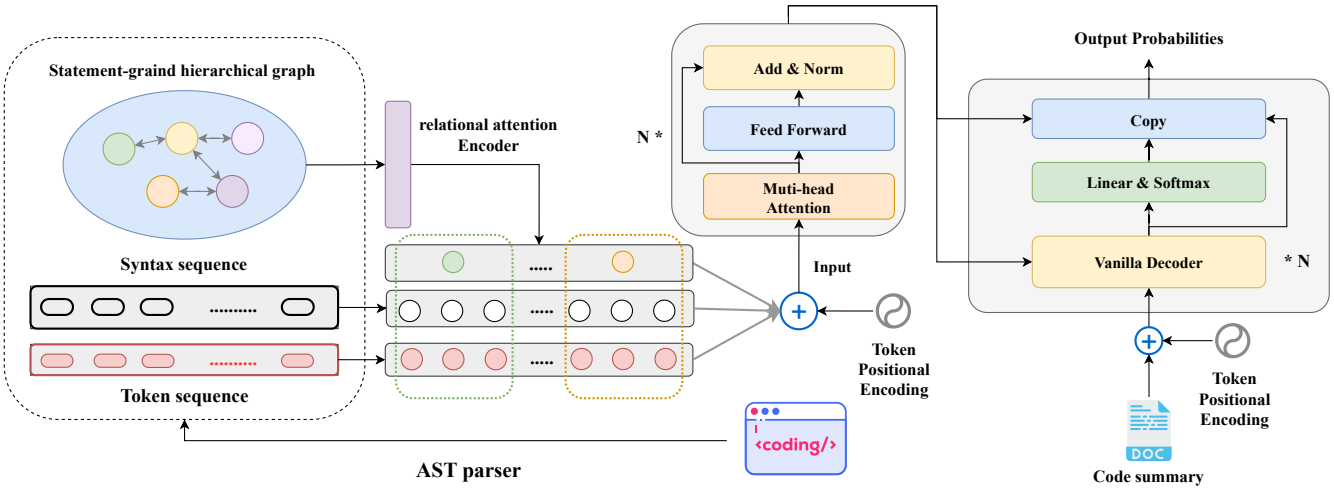


Figure 3. Model architecture of our approach.

4.1. Data Processing

According to [25], entities with distinct granularities highlight specific aspects of the source code from multiple perspectives. The statement-grained hierarchical graph reveals the code's global logical semantics and structural information. Similarly, the syntax sequence provides an in-depth analysis of the code's grammatical structure. Moreover, decomposing tokens into sub-tokens yields a textual analysis. For the extraction of these diverse granularities, each Java method is initially parsed into ASTs. ASTs are crucial as they encapsulate comprehensive syntax and semantic details. Our evaluation, focusing on Java programs, employs the *javaLang* parser, a Python library for Java code parsing. The following sections elaborate on the approaches used to construct the model input.

4.1.1. Construction of Statement-Grained Hierarchical Graph

Given a code snippet, the code is first parsed to obtain the AST, followed by a pre-order traversal of this AST. The *javaLang* parser encompasses a limited set of statement node types (*i.e.*, *methodDeclaration*, *statement*, *ifStatement*). When encountering an AST node whose tag is within the predefined set of statement node types, we collect the node into a set N. To represent the hierarchical structure of statements in a program, we introduce two types of hierarchical edges. A *sibling edge* connects statement nodes with the same parent node, whereas a *parent–child edge* links nodes in a parent–child relationship. Conversely, a *child–parent edge* represents a child–parent relationship. Formally, we define a semantic-grained hierarchical graph $G(N, E)$. For the code depicted in Figure 4, N consists of [*declaration*, *local_variable_declaration*, *while_statement*, . . . , *return_statement*], and E comprises {*parent–child edge*, *child–parent edge*, *sibling edge*}. The statement types elucidate the syntactic

information, while the hierarchical edges capture the global hierarchical information of the source code.

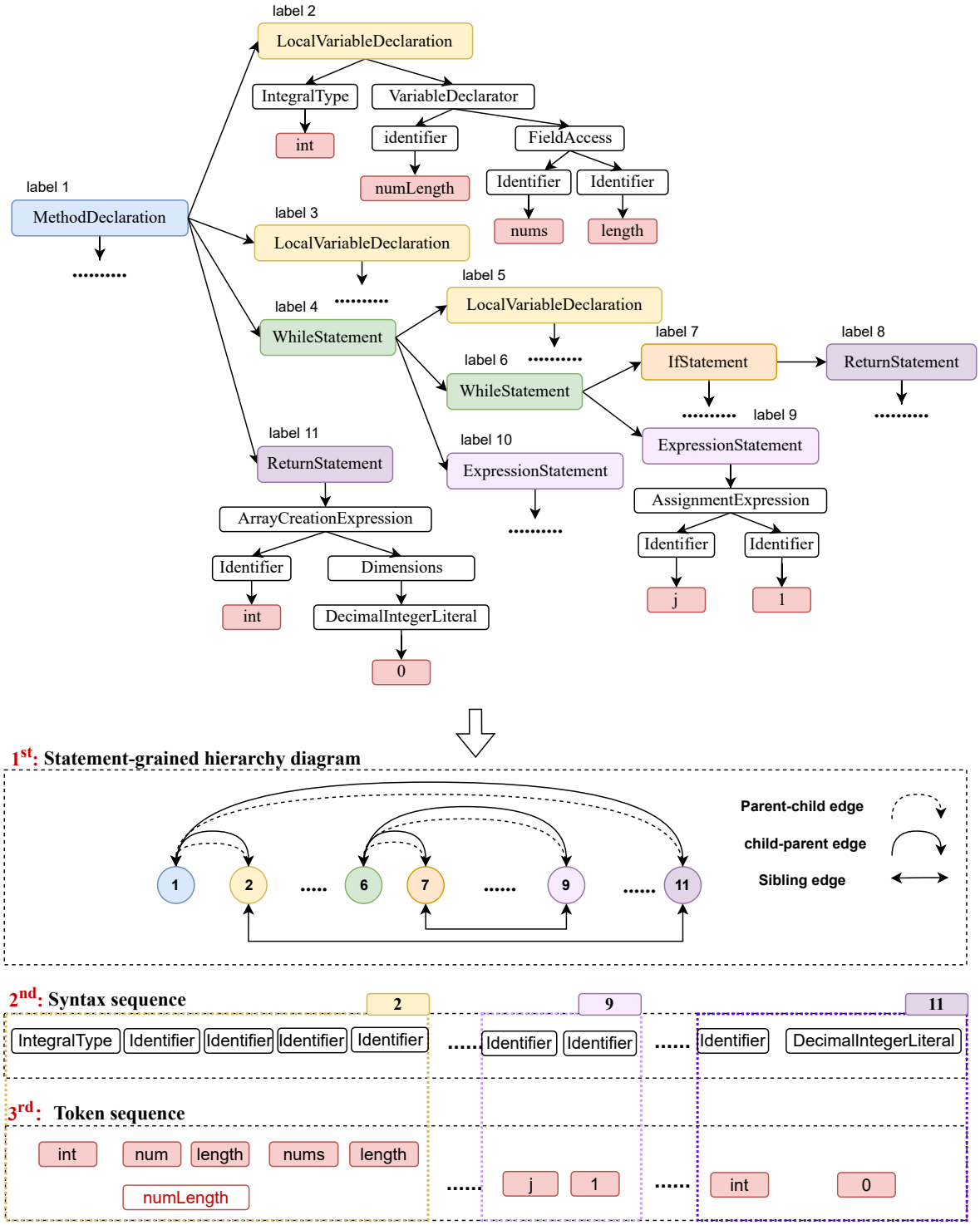


Figure 4. An example illustrating the construction of a statement-grained hierarchy graph, syntax sequence, and token sequence from AST.

4.1.2. Construction of Token and Syntax Sequences

Each leaf node of the AST corresponds to a specific token in the source code. Recognizing that the method and variable names in programs often comprise multiple natural language words, formatted in either camelCase or snake_case, we further decompose these

tokens into sub-tokens to reveal the textual feature of the program. The construction of of the sub-token sequence commences with the acquisition of the leaf nodes' sequence from the AST through depth-first pre-order traversal, followed by the division of each token into its constituent sub-tokens. The sub-tokenization approach, widely adopted in source code summarization methodologies [6,26], has been validated for its efficacy. Formally, we denote the ordered sub-token sequence as $T = [t_1, t_2, \ldots, t_n]$. In Figure 4, T is exemplified as [*public, int, two, To, Target, ...*]. The attribute of each token's parent node within the AST conveys its grammatical characteristic. To acquire the syntactical features of tokens, we generate a sequence of syntax nodes, wherein each token in T is substituted by its corresponding syntactical node (i.e., parent node). This sequence is formally represented as $S = [s_1, \ldots, s_n]$, indicating the ordered array of token syntactical types. As illustrated in Figure 4, S comprises elements like [*modifier, type, identifier, identifier, ...*]. Importantly, we ensure that the lengths of the sub-token sequence and the syntax sequence are aligned. Furthermore, we establish a one-to-many mapping relationship between statement nodes and token nodes, and this relationship is formally expressed as $M_{N \to T}$. When $M_{N \to T}(n_j, t_i) = 1$, it indicates that a token node t_i is derived from a statement node n_j.

In this study, we present three distinct code representations, each capturing different levels of granularity in program analysis: (1) a statement-grained hierarchical graph G, which focuses on the structure of the program by delineating the hierarchical relationships of statements; (2) a sub-token sequence T, offering detailed insights into the granular elements of the code, such as variable names and function calls; and (3) a syntax sequence S that highlights the grammatical features within the code. Collectively, these representations, $\{N, T, S\}$ provide a comprehensive multi-grained analysis of code, encompassing aspects from the overarching program architecture to fine-grained syntactic details.

4.2. Statement-Grained Hierarchy Transformer

We introduce the statement-grained hierarchy Transformer SHT, which integrates hierarchical, syntactical, and sequential textual information for code semantic representation. SHT follows the standard Transformer architecture but incorporates a two-phase encoder module: a relational attention-based hierarchical graph encoder and a sequence encoder. The first encoder employs relational attention to process the statement node's hidden states within the graph G, learning the statement-grained hierarchical graph's characteristics from two aspects: the global hierarchical structure and statement types. In the second phase, the sequence encoder is adopted to process the sequence representation, effectively aggregating hierarchical, syntactical, and textual information. This aggregation is crucial in deriving a final, enriched code representation that encapsulates multiple facets of the source code. Finally, the output from the encoder module is fed into a Transformer decoder for code summarization tasks.

4.2.1. Relational Attention-Based Hierarchical Graph Encoder

For Transformer-based architectures, Shaw et al. [23] show that relation features can be incorporated directly into the attention function by changing the attention computation. We leverage the relational attention encoder in the same form as [15] to encode the statement-grained hierarchical graph. Relational attention is built on the standard self-attention with relative position embeddings, but we replace the relative position embeddings derived from the linear relationship with the hierarchical edge types among statement nodes. For the statement-grained hierarchical graph $G = (N, E)$, we first embed the statement nodes and edge types, and then feed them into the relational attention encoder.

$$x_1, x_2, \ldots, x_l = Embed(n_1, n_2, \ldots, n_l) \tag{2}$$

The input of the relational attention encoder is a sequence $X = (x_1, x_2, \ldots, x_l)$, where x_i is a vector in $\mathbb{R}^d$ and represents the learned embedding of node n_i. Here, l denotes the length of sequence X. Additionally, e_{ij} symbolizes the learned embedding of the edge type that connects nodes i and j in the graph. It is important to note that if two nodes i and j

are not connected by hierarchical edges, then their corresponding edge e_{ij} is set to zero. $O = \{o_1, o_2, \ldots, o_l\}(o_i \in \mathbb{R}^d)$ is a sequence of output vectors of node N, which contains each node's global hierarchical information in the program. Relational attention can be formulated as

$$o_i = \sum_{j=1}^{n} \sigma(a_{ij}) V(x_i) \tag{3}$$

$$\alpha_{ij} = \frac{(Q(x_i) + \boldsymbol{e}(W_e)^T) K(x_j)^T}{\sqrt{d}} \tag{4}$$

where $Q, K : \mathbb{R}^d \to \mathbb{R}^m$ are query and key functions, respectively; $W_e \in \mathbb{R}^d$ is the projection matrix for edge embedding e_{ij}; $V : \mathbb{R}^d \to \mathbb{R}^d$ is a value function; and σ is a scoring function (e.g., softmax or hardmax). The $\sqrt{d}$ term is used to scale the attention weights.

4.2.2. Sequence Encoder

The sequence encoder is designed to encode a combination of hierarchical, syntactical, and textual representations of the code. The sequence of token embedding $[s'_1, s'_2, \ldots, s'_n]$ represents the textual features of the code snippet. To enhance the understanding of tokens, the sequence encoder also incorporates a sequence of syntax embedding $[t'_1, t'_2, \ldots, t'_n]$, which contains the syntactical information corresponding to the token. Additionally, the encoder stores a position embedding P, which indicates the absolute position of the tokens. The global hierarchical information among statement nodes also affects the tokens of each statement, so we utilize the one-to-many mapping relationship $M_{N \to T}(n_j, t_i)$ to pass the global hierarchical information into the tokens. Finally, the input of sequence encoder $[z_1, z_2, \ldots, z_n]$ is considered as the comprehensive representation of the code snippets, which is the sum of various embeddings, including tokens, syntax, absolute positions, and hierarchical information. The output of the sequence encoder $[h_1, h_2, \ldots, h_n]$ is passed to the decoder for summary generation.

$$s'_1, s'_2, \ldots, s'_n = Embed(s_1, s_2, \ldots, s_n) \tag{5}$$

$$t'_1, t'_2, \ldots, t'_n = Embed(t_1, t_2, \ldots, t_n) \tag{6}$$

$$z_i = s'_i + t'_i + \sum_{j=1}^{l} M_{N \to T}(n_j, t_i) o_j + p_i \tag{7}$$

$$h_1, h_2, \ldots, h_n = Transformerseq(z_1, z_2, \ldots, z_n) \tag{8}$$

4.2.3. Decoder with Copy Machine

In this work, we enhance the Transformer-based models by incorporating a pointer network [27] into the decoder. This integration significantly improves the prediction capabilities by allowing direct references to positions in the input sequence. As indicated by [26], approximately one third of tokens, in summary, are directly copied from the source code in code summarization datasets. This method proves particularly advantageous for less frequent tokens, facilitating the model's ability to predict out-of-vocabulary (OOV) tokens. Specifically, a decoder input h_i and decoder output v_i at timestep t are computed for a context vector h^*_t, as shown in Equation (9).

$$h^*_t = \sum_i softmax(W_1^T tanh(W_2 h_i + W_3 v_i + b_1)) h_i \tag{9}$$

The pointer generator operates at each timestep t by leveraging the context vector h^*_t, decoder input h_i, and decoder output v_i to calculate the generation probability $p_{gen} \in [0, 1]$. This probability serves as a soft switch, determining whether to generate a token from the model's vocabulary or to copy a token directly from the input sequence. The probability of predicting the token w at timestep t is calculated as follows:

$$P_{gen} = sigmoid(W_4 h_t^* + b_2) \tag{10}$$

$$p_{vocab} = softmax(W_5 h_t^* + b_3) \tag{11}$$

$$P(w) = (1 - p_{gen}) \sum_{(i:w_i=w)} a_i^t + p_{gen} P_{vocab} \tag{12}$$

where W and b are learnable parameters. p_{vocab} represents the probability distribution over all words in the vocabulary.

5. Experiment

In this section, we begin by presenting our experimental setup, which encompasses the evaluation datasets, metrics, comparison baselines, and parameter configurations. Then, the main results are reported, including a comparison with baselines, ablation studies, human evaluation, and qualitative analysis.

5.1. Experiment Setup

Evaluation Datasets

To evaluate the performance of the proposed method, we conducted our experiments using two public Java datasets: TL-CodeSum (https://github.com/xinghu/TL-CodeSum, accessed on 17 September 2022) [28] and EMSE-Deepcom (https://github.com/xing-hu/EMSE-DeepCom, accessed on 20 September 2022) [29]. These have been extensively utilized in prior research [6,25,26]. Specifically, the TL-CodeSum Java dataset comprises 87,136 pairs of Java methods and corresponding summaries, collected from GitHub repositories. The EMSE-Deepcom dataset contains approximately 485,812 pairs of Java methods and comments. To ensure consistency in dataset preprocessing and facilitate fair comparisons with the baseline models, we directly employed tokenized versions of these datasets, as provided by [6,25]. Both datasets were tokenized using the CamelCase and snake_case tokenizers to obtain sub-tokens [6], which can reduce the vocabulary significantly. For the summary sequences, we extracted the first sentence of Javadoc as the natural language description in the TL-CodeSum, while, in the EMSE-Deepcom dataset, we selected the first line of Javadoc as the corresponding summary. The original datasets do not include ASTs; therefore, we generated the associated ASTs from the raw source code using the javalang library (https://pypi.org/project/javalang, accessed on 17 September 2022), discarding unparsable methods.

It is important to note that some summaries were of limited utility. To maintain the data quality, we excluded comments with fewer than 4 words, as well as constructors, setters, getters, and tester methods, following the approach of [8]. We followed [25] to remove the deduplicated pairs of EMSE-DeepCom. Consequently, we obtained a final dataset comprising 83,661 pairs of source code and summaries for TL-CodeSum and 308,193 pairs for EMSE-DeepCom, respectively. Table 1 provides a breakdown of these datasets into training, validation, and test sets after applying the aforementioned filtering criteria. Additionally, we pre-shuffled all datasets to mitigate any potential impact resulting from the order of the data.

Table 1. Dataset statistics. #train indicates the number of instances for training. #/Avg.# tokens in code indicates the number of unique tokens and average method length in the code. The other terms are similar.

Dataset	TL-CodeSum	EMSE-Deepcom
#train	66,928	283,741
#validation	8366	12,226
#test	8366	12,226
#/Avg.# tokens in code	20,162/120.10	47,939/93.93
#/Avg.# tokens in summary	25,619/17.76	26,145/11.24

5.2. Metrics

We assess the source code summarization performance using three widely accepted metrics: BLEU [30], METEOR [31], and ROUGE-L [32]. These evaluation criteria are well established in the field of NLP and software engineering for the quality assessment of text generation [33,34]. This combination of metrics enables a comprehensive and well-rounded evaluation of our model's performance.

- BLEU calculates the n-gram precision overlap between two texts, serving as an accuracy measure. It indicates the proportion of generated text that corresponds to the reference text.
- METEOR, a recall-oriented metric, reflects the percentage of correctly generated content in comparison to the reference summary.
- ROUGE-L quantifies the longest common subsequence (LCS) between the reference and the generated code summary. It serves as a recall metric and provides additional insights not captured by the BLEU scores alone.

5.3. Comparison Baselines

To evaluate the performance of SHT, we conducted a comparative analysis with several well-established code summarization models serving as baselines.

- **CODE-NN** [5]: the first data-driven source code summarization model. It views source code as a sequential text and utilizes LSTM, a sequential model for the generation of source code summaries.
- **HDeepCom** [29]: a neural machine translation (NMT)-based code summarization model that converts the AST into a sequence by employing a structure-based traversal (SBT) method. It then feeds this sequence into a sequence-to-sequence model for comment generation.
- **ASTattGRU** [35]: this is a dual learning framework that jointly trains code summarization and code generation tasks, aiming to enhance both aspects simultaneously.
- **NeuralCodeSum** [6]: this is a Transformer-based code summarization model that leverages relative position information among code tokens to enhance the summary generation process.
- **CAST** [26]: this is a hybrid code summarization model that combines a tree-based Recursive Variational Neural Network (RvNN) and a basic code token encoder to capture both the code structure and sequence. It incorporates a hybrid mechanism in the decoding phase to combine inputs for the generation of descriptive summaries.
- **TPTrans** [36]: a Transformer-based code summarization model that captures the pairwise path information in the AST and integrates path encodings in the Transformer for concise summary generation.

We selected these baselines to establish comprehensive benchmarks for the performance evaluation of our approach for code summarization. During implementation, we either replicated the results precisely as reported in corresponding papers or rigorously reproduced the results using the publicly available repositories for most of the baselines.

5.4. Parameter Configurations

We followed the approach outlined by [25] to set the maximum sequence lengths and vocabulary sizes for both the code and summaries in both datasets. We constrained the vocabulary size to 50,000 for code sequences and 30,000 for summary sequences in both datasets. The maximum lengths of the code and summary were set to be 150 and 30. Meanwhile, the vocabulary size and sequence length of the AST were set as 50,000 and 200, respectively.

Our model architecture employed an embedding dimension and hidden size of 256, with 8 heads in each Transformer layer. The hierarchical encoder consisted of 2 layers, the sequential encoder had 4 layers, and the decoder comprised 2 layers. We trained the Transformer models using the Adam optimizer with an initial learning rate of 10^{-4}. We incorporated dropout with a probability of $p = 0.2$. The mini-batch size was set to 32. Our

experiments were conducted on the Pytorch 1.9.0 framework with an NVIDIA 3090 GPU. The training was capped at a maximum of 200 epochs, and we selected the checkpoint with the best performance on the validation set for subsequent evaluation on the test set. Additionally, to prevent overfitting, we implemented early stopping if the model's performance on the validation set did not improve for 20 epochs.

5.5. Main Results

To provide a comprehensive analysis of our model's performance, we begin by conducting a comprehensive performance comparison with the baselines. Additionally, we perform an ablation study to assess the effectiveness of our proposed code semantic representation. Furthermore, we perform human evaluations to assess the generated code summaries regarding informativeness and readability. Finally, we present examples of generated summaries to qualitatively illustrate the results.

5.5.1. Comparisons with Baselines

The overall results of our proposed model and the baselines are presented in Table 2. Our model outperforms all the baselines, showing robust generalization performance on both datasets. CodeNN, employing exclusively code tokens with an LSTM network, captures sequential token information. In comparison, HDeepCom and ASTattGRU encode AST structures using GRU and then concatenate this encoding with the token sequence to predict the summary. Notably, CodeNN achieves higher METEOR scores and lower BLEU scores than HDeepCom, primarily attributed to its tendency to generate shorter summaries. This observation highlights the impact of the code structure on code comprehension and summary generation, as both HDeepCom and ASTattGRU outperform CodeNN. NeuralCodeSum exclusively encodes code tokens, achieving superior performance by utilizing a Transformer to capture long-range dependencies among code tokens. Moreover, the Transformer-based approaches, CAST and TPTrans, outperform LSTM-based methods in code summarization tasks. Compared to NeuralCodeSum, which integrates pairwise relative positional information among tokens into the encoder, our approach utilizes relational attention encoding to capture global hierarchical information among statements, yielding notable improvements in the BLEU, ROUGE-L, and METEOR scores, with increases of 5.13% and 1.72%, 3.7% and 1.15%, and 3.37% and 0.85% on both datasets, respectively. In comparison to the leading methods, CAST and TPTrans, which take all nodes in the ASTs and token sequences as input, our model, which aggregates global hierarchical features, syntax, and tokens, demonstrates superior performance on both datasets. These results validate the rationale behind our approach, which combines statement-level hierarchical information with conventional syntax and token textual information, as an effective strategy for code summarization.

Table 2. Comparisons with baseline methods on both TL-CodeSum and EMSE-DeepCom datasets.

Approaches	Input	Backbone	TL-CodeSum			EMSE-Deepcom		
			BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
CodeNN [5]	Code	LSTM	22.22	33.14	14.08	28.45	43.51	17.89
HDeepcom [29]	AST	GRU	23.32	33.94	13.76	32.19	49.03	21.53
ASTattGRU [35]	AST	GRU	30.78	39.94	17.35	33.40	49.76	22.20
NeuralCodeSum [6]	Code	Transformer	40.63	52.00	24.85	37.13	54.87	25.05
CAST [26]	AST	Transformer	43.76	54.09	27.15	37.19	54.87	25.07
TPTrans [36]	AST	Transformer	44.50	55.08	27.88	37.25	54.99	25.02
SHT	AST	Transformer	45.76	55.70	28.22	38.85	56.02	25.90

5.5.2. Ablation Study

We further conducted ablation studies to evaluate the impact of various code structural properties, hierarchical granularity, and aggregation strategies. Initially, to assess the

influence of different code structure types, we individually omitted the code tokens and token syntax. Additionally, we transferred the hierarchical relationships from statement nodes to the individual tokens that they encompassed to evaluate the effect of hierarchical granularity on code semantic comprehension. Finally, we explored different strategies for the aggregation of the multi-structural properties of the source code. These ablation experiments were carried out on the TL-CodeSum dataset, with the training setup unchanged unless specified otherwise. The results are illustrated in Table 3.

Table 3. Evaluation results of ablation study on TL-CodeSum dataset.

Approach	BLEU	ROUGE	METEOR
SHT	45.76	55.70	28.22
w/o token	42.33	53.27	26.89
w/o syntax	45.38	55.03	27.63
token-grained hierarchy	45.08	55.17	27.90
concatenation	45.42	55.36	28.06

The performance comparison, omitting either syntax embeddings or token embeddings, highlights the greater significance of token textual information over token syntax in code semantic representation. Remarkably, removing tokens led to an approximately 8.1% decline in the BLEU score, whereas omitting syntax resulted in a small decrease of about 0.8%. The result suggests that the modeling of semantic relations among tokens is beneficial for the code summarization task. Concurrently, the preservation of both the textual and syntactic features of tokens enables a more comprehensive representation of the source code's semantic information.

Transferring the global hierarchical relationships from statement nodes to tokens substantially increases the complexity of the token-grained hierarchical graph. Furthermore, given the relatively minor impact of the tokens' syntax on code semantic representation, we opted to employ only the relational attention encoder to learn the token-grained hierarchical graph for summary prediction. As indicated in Table 3, this approach of capturing code semantic information from a token-grained hierarchical graph resulted in a performance decrease of approximately 1.4%. This suggests that token-grained hierarchical graphs may struggle to represent the global hierarchy within code snippets effectively. It is noteworthy that statements, as fundamental units to convey source code semantics, also represent an ideal level of granularity for the encapsulation of the global hierarchy.

Regarding the input of our sequence Transformer, which includes the token's textual embedding, syntax embeddings, and hierarchical features, several aggregation strategies are feasible. In our model, we chose a straightforward approach of directly adding these elements together for summary prediction. Ref. [37] describes various strategies for multi-source Transformers. For simplicity, our contrast experiment employed concatenation instead of component-wise addition, replacing $z_i = s_i + t_i + o_i$ with $z_i = [s_i; t_i; o_i]$. As demonstrated in Table 3, the concatenation strategy did not perform as well as the simple addition of multi-granularities. Nevertheless, it delivers comparable performance, with the concatenation strategy achieving a BLEU score of 45.08, which is not significantly different from that of our original strategy.

5.5.3. Human Evaluation

To further assess the quality of the code summaries generated by our proposed model, we conducted a human evaluation, comparing its generative capabilities against three state-of-the-art baselines. We randomly selected 50 samples for evaluation, comprising 25 from the test sets of TL-CodeSum and 25 from EMSE-CodeSum, respectively. Five experienced developers, each boasting over three years of software development experience and proficient English skills, were invited to participate in this evaluation. The developers were tasked with assessing the generated summaries based on three criteria, *similarity*, *conciseness*, and *readability*, rating them on a scale from 1 (very dissatisfied) to 5 (very satisfied).

Similarity refers to how closely the generated summary matches the ground truth summary. *Conciseness* evaluates the extent to which the summaries describe the code's functionality without unnecessary or extraneous information. *Readability* focuses on the grammatical correctness and fluency of the generated code descriptions. The results of this evaluation are presented in Table 4. The final score for each example represents the average of the scores given by the five developers. The findings from the evaluation reveal that the summaries produced by our model surpass all baselines across all evaluated metrics, indicating the efficacy of our approach in generating summaries with comprehensive semantics.

Table 4. Human evaluation results.

Approach	Similarity	Conciseness	Readability
SHT	3.52	3.17	3.34
NeuralCodeSum	2.51	2.72	2.34
CAST	3.11	2.90	2.97
TPTrans	3.20	3.17	3.34

5.5.4. Qualitative Analysis

To visually demonstrate our model's performance, we present two examples for qualitative analysis, as depicted in Table 5. These examples illustrate that our proposed model not only generates summaries closely resembling the reference but also maintains readability. We specifically focused on the summary generation capabilities of our approach versus the token-grained hierarchy model. In the first example, a code snippet contains a simple structure with three statement types, characterized primarily by parent–child relationships. The token-grained hierarchy model, in this case, produces summaries with certain semantic relevance but less readability. The second example involves a more complex hierarchical structure. Here, the token-grained hierarchy model generates summaries with a limited understanding of the source code's semantics. However, despite ignoring the syntax, the token-grained hierarchy model still manages to grasp the core function in its summaries. This suggests that the hierarchical information of source code excels in capturing global structural and logical relationships for code comprehension. The results of these qualitative examples indicate that our approach, which considers the global hierarchy, syntax, and text of the source code, is adept at generating concise and readable summaries for both simple and complex code snippets.

Table 5. Qualitative examples.

Source Code

```
private void sendRemainingParts(Client client, String[]
    strings){
     for(int i = NUM_; i < strings.length; ++i){
          client.appendMessage(strings[i]);
          }
     }
```

Summaries:
Reference: send the remaining parts of a string array to a client
SHT: send the remaining elements of a string array to a client
SHT w/o syntax: Sends array remaining strings to client
token-grained hierarchy: send remaining string array to a client

Table 5. *Cont.*

Source Code

```
public boolean containsValue(Object value){
        Entry tab[] = table;
        if(value == null){
                for(int i = tab. length; i -- > NUM_;)
                for(Entry e = tab[i]; e! = null; e = e.next)
                if(e.value == null) return BOOL_;
        }
        else {
                for(int i = tab.length; i -- > NUM_;)
                for(Entry e = tab[i]; e! = null; e = e.next)
                if(value.equals(e.value))
                return BOOL_;
        }
        return BOOL_;
    }
```

Summaries:
Reference: check whether a given value exists in a collection of objects
SHT: check whether a given value exists in a collection of objects
SHT w/o syntax: search entries for value, returns predefined value
token-grained hierarchy: search entries for value, returns predefined value

6. Related Work

Research on source code summarization has been extensively conducted for over a decade, and it can be categorized into three primary types: template-based approaches, information retrieval (IR) based approaches, and data-driven approaches. With the rapid advancement of deep learning techniques, data-driven approaches have become the predominant method, achieving the best performance in recent years. Consequently, in this section, we mainly focus on data-driven approaches in related works. Source code summarization is considered a sequence generation task and adopts Natural Machine Translation (NMT) frameworks with encoder-decoder models to enhance the performance.

Most approaches either treat the source code as a sequence of tokens or a structural representation from an AST or graph (i.e., control flow, program dependency), thereby employing various encoders to handle the diversity of inputs. To encode the code sequence, Iyer et al. [5] were the first to propose an RNN-based model with a Long Short-Term Memory (LSTM) network. Ahmad et al. [6] utilized a Transformer model with relative position embedding to capture the code's semantic representation. For the extraction of structural information from ASTs, the Tree-LSTM [1,8] and Tree-Transformer [9] models adopt RNN-based and Transformer-based frameworks, respectively, to directly encode ASTs. Additionally, some approaches have worked with variants of ASTs. Hu et al. [10] proposed the "Structure-Based Traversal" (SBT) method to flatten ASTs, leveraging an attention-based RNN encoder to encode flattened ASTs via the Structure-Based Traversal (SBT) method for code summarization. Alon et al. [11] extracted paths from an AST and encoded these paths using random walks. CAST [26] leverages a tree-based Recursive Neural Network to encode reconstructed subtrees of ASTs. Graph-based methods represent the program as a graph in acquiring a deeper code semantic comprehension [12]. Fernandes et al. [13] and Junyan et al.[14] encoded graph-structured representations by using a Graph Neural Network (GNN). To capitalize on the advantages of both the code sequence and structural information, some researchers have attempted to jointly learn these aspects for code representation. GREAT [15] utilizes the Transformer model to encode an AST-based graph, which adds edges of data flow and control dependency into the tokens of the

AST; furthermore, relative positional encoding is utilized to acquire the pairwise relations between tokens. CodeTransformer [16] also considers the distance on the AST-based graph in the self-attention operation.

7. Limitations

There are three main threats to the validity of our study.

- Dataset limitations: While numerous public datasets are available for the code summarization task, our model evaluation was confined to only two public Java datasets. Consequently, these datasets may not fully represent other programming languages, potentially limiting the scalability of our model. In future work, we plan to experiment with more large-scale datasets encompassing diverse programming languages. We anticipate that our model could be extended to other languages capable of being parsed into ASTs with minimal adaptation.
- Hyperparameter settings in deep learning: The configuration of the dimensions plays a pivotal role in influencing the outcomes of a deep learning model. We conducted a limited-range grid search focusing on the learning rate and batch size to optimize our model's performance. To mitigate the impact of varying hyperparameter settings among baseline models, we compared our performance against the best results reported in previous works for these baselines.
- Biases of human evaluation: We incorporated human evaluation by inviting five participants to assess the quality of 50 code–summary pairs, selected randomly. It is important to acknowledge that the outcomes of human annotations can be influenced by various factors, including the participants' programming experience and their comprehension of the evaluation criteria. Recognizing this potential for bias, future iterations of our study will seek to involve a larger number of skilled software developers for the evaluation of an expanded set of code–summary pairs. Additionally, to further enhance the reliability of our findings, each code–summary pair will be reviewed by a minimum of five participants.

8. Conclusions

In this article, we introduce SHT, a Transformer-based architecture designed to generate summaries for Java methods. This model uniquely integrates the globally hierarchical and sequential information of source code snippets. In SHT, source code is parsed from three perspectives: a statement-grained hierarchical graph and sequences of syntax and tokens. The hierarchical graph is encoded using a relational attention encoder, after which the hierarchical features are combined with tokenized sequences by another sequence encoder. This process facilitates the effective extraction of code semantics. Our comprehensive experiments demonstrate that SHT achieves state-of-the-art results on two public Java benchmarks. Looking ahead, we plan to expand our approach to other programming languages using larger datasets. Moreover, we aim to adapt and evaluate our model across various coding task domains. This expansion is anticipated to further enhance the accuracy of code semantic representation, broadening its applicability and utility in the field of software engineering.

Author Contributions: Conceptualization, Q.Z.; methodology, Q.Z.; software, Q.Z.; validation, Y.W. and D.J.; investigation, Q.Z.; resources, Y.G.; writing—original draft preparation, Q.Z.; writing—review and editing, Q.Z and D.J; supervision, Y.W.; project administration, Y.W. and D.J.; funding acquisition, Y.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (No. U1736110).

Data Availability Statement: Due to privacy issues, we are unable to publish the dataset.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Wan, Y.; Zhao, Z.; Yang, M.; Xu, G.; Ying, H.; Wu, J.; Yu, P.S. Improving automatic source code summarization via deep reinforcement learning. In Proceedings of the Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 397–407.
2. Xia, X.; Bao, L.; Lo, D.; Xing, Z.; Hassan, A.E.; Li, S. Measuring program comprehension: A large-scale field study with professionals. *IEEE Trans. Softw. Eng.* **2017**, *44*, 951–976. [CrossRef]
3. Stapleton, S.; Gambhir, Y.; LeClair, A.; Eberhart, Z.; Weimer, W.; Leach, K.; Huang, Y. A human study of comprehension and code summarization. In Proceedings of the 28th International Conference on Program Comprehension, Seoul, Republic of Korea, 13–15 July 2020; pp. 2–13.
4. Liu, S.; Chen, Y.; Xie, X.; Siow, J.; Liu, Y. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv* **2020**, arXiv:2006.05405.
5. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing source code using a neural attention model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; Association for Computational Linguistics: Stroudsburg, PA, USA, 2016; pp. 2073–2083.
6. Ahmad, W.U.; Chakraborty, S.; Ray, B.; Chang, K.W. A transformer-based approach for source code summarization. *arXiv* **2020**, arXiv:2005.00653.
7. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. CSUR* **2018**, *51*, 1–37. [CrossRef]
8. Shido, Y.; Kobayashi, Y.; Yamamoto, A.; Miyamoto, A.; Matsumura, T. Automatic source code summarization with extended tree-lstm. In Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN), Budapest, Hungary, 14–19 July 2019; pp. 1–8.
9. Harer, J.; Reale, C.; Chin, P. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv* **2019**, arXiv:1908.00449.
10. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension, Gothenburg, Sweden, 27 May–3 June 2018; pp. 200–210.
11. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating sequences from structured representations of code. *arXiv* **2018**, arXiv:1808.01400.
12. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to represent programs with graphs. *arXiv* **2017**, arXiv:1711.00740.
13. Fernandes, P.; Allamanis, M.; Brockschmidt, M. Structured neural summarization. *arXiv* **2018**, arXiv:1811.01824.
14. Cheng, J.; Fostiropoulos, I.; Boehm, B. GN-Transformer: Fusing Sequence and Graph Representation for Improved Code Summarization. *arXiv* **2021**, arXiv:2111.08874.
15. Hellendoorn, V.J.; Sutton, C.; Singh, R.; Maniatis, P.; Bieber, D. Global relational models of source code. In Proceedings of the International Conference on Learning Representations, New Orleans, LO, USA, 6–9 May 2019.
16. Zügner, D.; Kirschstein, T.; Catasta, M.; Leskovec, J.; Günnemann, S. Language-agnostic representation learning of source code from structure and context. *arXiv* **2021**, arXiv:2103.11318.
17. Zhang, K.; Li, Z.; Jin, Z.; Li, G. Implant Global and Local Hierarchy Information to Sequence based Code Representation Models. *arXiv* **2023**, arXiv:2303.07826.
18. Kitaev, N.; Klein, D. Constituency parsing with a self-attentive encoder. *arXiv* **2018**, arXiv:1805.01052.
19. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 1–11.
20. Song, K.; Wang, K.; Yu, H.; Zhang, Y.; Huang, Z.; Luo, W.; Duan, X.; Zhang, M. Alignment-enhanced transformer for constraining nmt with pre-specified translations. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 8886–8893.
21. Zhao, X.; Wang, L.; He, R.; Yang, T.; Chang, J.; Wang, R. Multiple knowledge syncretic transformer for natural dialogue generation. In Proceedings of the The Web Conference 2020, Taipei, Taiwan, 20–24 April 2020; pp. 752–762.
22. Tang, Z.; Shen, X.; Li, C.; Ge, J.; Huang, L.; Zhu, Z.; Luo, B. AST-trans: Code summarization with efficient tree-structured attention. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 150–162.
23. Shaw, P.; Uszkoreit, J.; Vaswani, A. Self-attention with relative position representations. *arXiv* **2018**, arXiv:1803.02155.
24. Diao, C.; Loynd, R. Relational attention: Generalizing transformers for graph-structured tasks. *arXiv* **2022**, arXiv:2210.05062.
25. Chai, L.; Li, M. Pyramid Attention For Source Code Summarization. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 20421–20433.
26. Shi, E.; Wang, Y.; Du, L.; Zhang, H.; Han, S.; Zhang, D.; Sun, H. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. *arXiv* **2021**, arXiv:2108.12987.
27. Vinyals, O.; Fortunato, M.; Jaitly, N. Pointer networks. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 1–9.
28. Hu, X.; Li, G.; Xia, X.; Lo, D.; Lu, S.; Jin, Z. Summarizing source code with transferred api knowledge. In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, Stockholm, Sweden, 13–19 July 2018.
29. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.* **2020**, *25*, 2179–2217. [CrossRef]
30. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.J. Bleu: A method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, Philadelphia, PA, USA, 6–12 July 2002; pp. 311–318.

31. Banerjee, S.; Lavie, A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, Ann Arbor, MI, USA, 29 June 2005; pp. 65–72.
32. Lin, C.Y. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2004; pp. 74–81.
33. Liu, Z.; Xia, X.; Treude, C.; Lo, D.; Li, S. Automatic generation of pull request descriptions. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 10–15 November 2019; pp. 176–188.
34. Nie, L.Y.; Gao, C.; Zhong, Z.; Lam, W.; Liu, Y.; Xu, Z. Coregen: Contextualized code representation learning for commit message generation. *Neurocomputing* **2021**, *459*, 97–107. [CrossRef]
35. LeClair, A.; Haque, S.; Wu, L.; McMillan, C. Improved code summarization via a graph neural network. In Proceedings of the 28th International Conference on Program Comprehension, Seoul, Republic of Korea, 13–15 July 2020; pp. 184–195.
36. Peng, H.; Li, G.; Wang, W.; Zhao, Y.; Jin, Z. Integrating tree path in transformer for code representation. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 9343–9354.
37. Libovickỳ, J.; Helcl, J.; Mareček, D. Input combination strategies for multi-source transformer decoder. *arXiv* **2018**, arXiv:1811.04716.